A 96-20

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Announcement 9/94 - 4/95

4. TITLE AND SUBTITLE
Mapping the Method Muddle: Guidance in Using Methods for Used Interface Design

5. FUNDING NUMBERS
PE: 0601102A
PR: B74F
TA: 1901
WU: C19 MDA
Contract No. 903-89-K-0025

6. AUTHOR(S)
Judith S. Olson (University of Michigan) and Thomas P. Moran (Xerox PARC)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Michigan
Division of Research and Development
475 East Jefferson, Room 1318
Ann Arbor, MI 48109 - 1248

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U. S. Army Research Institute
ATTN: PERI-BR
5001 Eisenhower Avenue
Alexandria, VA 22333-5600

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
Announcement 96-20

11. SUPPLEMENTARY NOTES
COR: Dr. Michael Drillings In C. Lewis, Tim McKay, P. Polson, & M. Rudisill (EDS.) Human-computer interface design: Success cases, emerging methods, and real world contexts. New York: Morgan Kaufman.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release; Distribution is Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*: This overview shows that there is considerable progress in providing ways to design useful, usable, and learnable user interfaces. Many new methods have been developed since the 1983 NRC report, and recent studies have compared the cost/benefit of various methods. We have provided a framework for seeing the roles of different methods, but more work is needed on a detailed cost/benefit of the methods. Not only do the methods med to be assessed for their usefulness, but new methods need to be developed that are more complete and usable.

14. SUBJECT TERMS	15. NUMBER OF PAGES
interface designs utility prototype envisioning semantic nets user task	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Mapping the Method Muddle: Guidance in Using Methods for User Interface Design

Judith S. Olson
University of Michigan
and
Thomas P. Moran
Xerox PARC

September, 1994

A great deal of progress has been made in developing methods that designers can incorporate in appropriate places in the design process with the goal of producing interfaces that are easier to use and easier to learn. This book represents another step in this line, reporting on emerging methods for designers. However, we have not yet succeeded in making user-interface methods an established part of software design practice. Many designers claim they do not have enough time for usability. Even when designers *want* to design for usability, they find that the literature on design methods is a muddle, making it difficult to figure out if there are methods that are appropriate for their situations. They need to know what they can do, and when. The purpose of this paper is to provide that guidance.

In 1983, there was a similar effort. The National Research Council's (NRC) Committee on Human Factors published the proceedings of a workshop entitled, "Methods for Designing Software to Fit Human Needs and

Capabilities" (Anderson and Olson, 1985). Table 1 lists the methods included in that report. [1]

Table 1
Methods for User Interface Design listed in the National Research Council's 1983 Report.

Questionnaires and *interviews*, *diaries*, *natural observation* of task performance, *activity analysis*, *logging and metering* for understanding the old situation;

task analysis for understanding the requirements for the new task;

design guidelines, *user reactions*, and *theory based judgments* to assess the initial design, with support from *toolkits* to guide designers to select from some well established design pieces, such as dialog boxes preconfigured and menu bars into which to put command names;

formal analyses of interfaces using *structured walkthroughs*, *decomposition analysis*, *object/action analysis*, *metaphor analysis*, *mental model* assessment, *GOMS/keystroke analysis*, and *grammar analysis*;

building a prototype through *facading*, *Wizard of Oz*, or *rapid prototyping;*

using these prototypes in *usability tests;*

and, running the proposed final design in friendly *field tests* (similar to beta test sites for the system code).

Although the major methods used today are the same as those in the NRC report (Nielsen, 1993), there are some new additions. The collection of papers

[1]Of the participants in this book, many were also participants in the NRC Committee on Human Factors' workshop in 1983. Stu Card, Jack Carroll, Judy Olson, and John Whiteside were at both. Others in the 1983 workshop included: Nancy Anderson, Elizabeth Bailey, Alphonse Chapanis, Rex Hartson, David Lenorovitz, Marilyn Mantei, Dick Pew, Phyllis Reisner, Janet Walker, and Bob Williges.

in this book updates that list to include not only more methods, but also to illustrate their successful use in real design projects. Yet designers still need some help sorting through the methods in order to choose those applicable to the design situation at hand. This help is needed even more so now because the set of methods has grown, some are quite sophisticated, and they differ in applicability to the stage of design and to the kind of task being designed for.

To help orient designers, we begin by a characterization of the design process as consisting of several activities. Methods are then described in abstract form (there are too many to detail here) and associated within the appropriate activity. We then lay out a way for the designer to choose the methods appropriate to key features of the task and users in their design situation: the speed of performance expected of the user, the amount of information content in the task domain, and whether the focus is on rapid learning or expert performance. It is not expected that a designer will choose to use only one of these methods, but rather select a set of coordinated methods. We illustrate the use of the methods with scenarios of two contrasting design situations. We end with a discussion about the methods and recommendations for further development of design methods that carry with them the appropriate cost/benefit to the designers.

What is a Method?

We use the term "method" as meaning *how to go about designing something.* A method implies a systematic, repeatable way to design. There are macro- and micro-methods. A macro-method is a methodology that organizes the whole design process. Software engineering methodologies, like Jackson System Diagramming and Object Oriented Methods, tend to be macro. Large

companies usually advocate or even require formalized design methodologies to manage large-scale design projects. Micro-methods are methods that only address subproblems of design. An extended design process would employ several micro-methods, which would be used in combination as needed. In this paper, we discuss only micro-methods.

A complete method would include:

- a statement of the ***problem*** that this method addresses,
- a ***device*** (a tool, technique, model, and or representation),
- a ***procedure*** for using the device, and
- a ***result***.

For example, a method called "Cognitive Walkthrough" (Lewis, Polson, Wharton, and Rieman, 1990; Rieman, Davies, Hair, Esemplare, Polson, and Lewis, 1991) is complete. It addresses the ***problem*** of designing a system to be easy to use when the user first walks up to it. It has a set of forms for the designer to fill out (the ***device*** is a representation) that describe the user's steps in performing a task, and a ***procedure*** for "walking through" the steps, asking questions about how easily the user will discover how to do the next step. The ***result*** is a list of aspects of the user interface that are likely to give the user trouble, which guides the designer to areas of the interface that should be altered.

There are very few complete methods for user interface design. Usually, there is a technique or model or representation, but no explicit procedure for using it. Sometimes the problem addressed is vague, and sometimes the result is implicit. Often these missing aspects can be supplied by the designer in the

context of a particular design situation. Therefore, we will simply refer to "methods" in this paper, even if incomplete.

What Can Be Done to Improve Design

The key to good systems is the commitment of time and expertise to the user-oriented aspects of the design. The designers have to know more than computer science and technology; they need training and experience in making systems useful, usable and acceptable. Methods help guide this effort.

A system has ***utility*** when it meets users' needs and solves problems that users have. To accomplish this goal, we need to understand the user's practices and work setting, including both what their goals are and how they go about accomplishing those goals. The new system must provide appropriate functionality to meet the needs in the context of the work setting (i.e., fitting the organizational culture, the established communication patterns, and the incentive structure, among other things).

The functionality must be offered through a ***usable*** interface. The system must be understandable and learnable, relative to the skills of the users and the resources available for learning. The system must have the right performance characteristics: Users must be able to do the tasks fast enough and accurately enough to meet their needs.

Finally, the system must be subjectively ***acceptable*** to the users. They must perceive benefit from it and perhaps even enjoy using it. Such acceptance comes from both a well-designed system and a design process that

successfully targets the right market and involves users in the design and deployment of the system.

Design methods can help achieve better design. System design is an art which mixes creativity and discipline. Most of the methods listed here help with the discipline, not with the creativity. Of course, regular methods that help with brainstorming ideas and structuring and evaluating them apply to the design process (e.g., Adams, 1979). But, the design methods reviewed in this chapter are more specific and engineering-like. They:

- focus attention on users' needs and capabilities,
- provide tools to represent and build designs,
- encourage thorough thinking by systematizing design activities and bringing theory and knowledge to bear where it can be useful, and
- foster testing and measurement.

Focus attention on users needs and capabilities. The key feature of good design methods is that they focus on the user. One way to do this is to include users themselves in the design process. When new designs are proposed, users should be involved in testing them. Since users cannot always articulate what they know about a procedure or its goal, it is sometimes important to observe users in their current practice. Even better, users should participate in the creation of the new system, defining requirements, constructing prototypes, etc. Almost all methods discussed in this chapter can be enhanced by having users themselves involved.

Probably the most important step in designing good systems is getting the functionality right. It is most important to clearly understand the needs early

in the process. Design methods can help investigate the users' task domain and current work practices. One can distill important user scenarios, making clear what the critical criteria are. Most methods rely on having a set of tasks or scenarios. Choosing the right ones is important. It is useful to have a mix of *core tasks* (those most central or frequent), *critical tasks* (tasks which illustrate the new capabilities), and *benchmark tasks* (tasks that can be used to compare the new system with other systems).

Provide tools to represent and build designs. Design methods provide useful representations for design. Good representations are important for many reasons. Representations help us see a proposed design in a particular way and come to an understanding of it from that view. Concrete representations help with communication between members of the design team and between designers, users, and others stakeholders. Some representations, called *models*, allow calculations of properties of the design, such as the amount of new things to learn or the time the user would be expected to execute a task.

Prototyping is important, in that it provides an understanding of many issues at an early stage, when something can still be done to deal with them. Some design methods provide tools that make it easier to build prototypes and even final implementations.

Encourage thorough thinking and analysis. Often the greatest value of a representation comes in the effort to create it, which requires careful thinking through of particular issues. Reflecting is an integral part of design. Design methods can aid reflection both by providing representations that

"talk back" (Schon, 1983) and by disciplining the designer to be concrete and complete. Methods such as checklists and walkthroughs promote good reflection and thorough analysis of aspects of the design which might cause users difficulty.

Foster testing and measurement. We must never assume that designs will work as planned. They need to be tried and tested. Some design methods provide ways to test and measure prototypes and implementations. Methods codify other designers' experience by prescribing particular tests and measures that have been shown to be informative for discovering difficulties and pointing to areas that need redesign.

The Activities That Designers Engage In

System design is a complex process which varies from situation to situation. For example, designing a generic new product for which there is only a description of a market is quite different than designing a custom system for a specific user in a specific setting. The process must accommodate both these large-scale differences and the specifics of the tasks and settings within them. Nevertheless, there are many recurring activities in design, for which the methods reviewed in this chapter are intended to provide support. We here consider seven common activities:

Define the Problem. As we have stressed, the most important thing is to get the problem right. Design theorists (Simon, 1981) have described design as an *open problem* in that the problem is not given from the start, but is clarified as the solution is being formulated. Part of defining the

problem is to understand the task domain and current work practices of the user community. From this, the designer must figure out what the needs are that can be met with a new system. Defining the problem is not just an analytic endeavor; it opens up possibilities.

Generate a Design. Once there is some sense of the problem and at least a general notion of the kind of system needed, a more detailed design can be created. Generating a design is very much a creative and constructive activity, involving the building of concrete representations showing what the system would be like.

Reflect on the Design. Once any part of the design is represented, it has to be assessed. This is done by reflecting on it--living with it in the imagination, analyzing it, challenging it, and so forth. This leads to new ideas. Thus, generating and reflecting are tightly coupled activities.

Build a Prototype. A prototype is a concrete representation for which some aspects "work." It could be a physical mockup, a set of pictures, or running software that can portray the design in a way comprehensible by someone other than designers.

Test the Prototype. The point of a prototype is to "try it out" informally or to test it more systematically with potential users. The goal here is to discover fatal flaws, new issues, and aspects of the design that seem to work especially well.

Implement the Design. Building the real system involves a commitment to a lot of decisions, since the flexibility to alter the design rapidly becomes less practical. Following software engineering principles can help not only with efficient implementation, but also with modularity that will preserve some flexibility.

Deploy the System. As soon as the system implementation reaches a usable stage (well before it is complete), it can begin to be deployed in different ways. There are several strategies for trying it out in limited ways, both to learn if it will serve the intended needs and to get the user community prepared for its installation. One must also be conscious of the fact that once deployed, the task changes; this leads to the need to evaluate iteratively, and inform the next generation of products.

It cannot be stressed enough that these activities are **not** stages in the design process. The course of design involves continual jumping back and forth between the activities. There is iteration in which feedback from activities later in the list inform those earlier.

Although these activities are common in design, they do not define a complete design and development process. They are presented here to provide an organizational framework for the multitude of design methods reviewed here.

The Methods

The collection of methods for user-interface design have clear relevance for different stages in the design cycle. Some methods help the analyst at the

beginning of the project to understand the task and the setting in which the new system will reside. Other methods help generate the working design. Still others are intended for analysis of the proposed design, to help in suggesting improvements in learnability and usability. Methods in the listing below are clustered and ordered according to the seven activities described above.

Each section below begins with a summary table which lists the methods, several descriptors that should guide the designer in deciding whether it is useful, and references that describe its use in a design setting or show how to do it. The two descriptors we offer at this stage are estimates of how long it takes to do this method (called "effort"), and how much training is required in order to make the prescribed assessments. The training is assumed to consist of a short-course or tutorial (for example, from the conference called Computer Human Interaction, CHI) plus some exercises and apprenticeship. The person being trained is either a designer or human-factors person, typically someone with a bachelors degree. The estimate of effort is a rough guess of how long the analysis or method would take the person so trained. Both of these estimates come from the authors' experience in both using these methods as well as teaching these methods and observing their use in a variety of settings. [2] Although many contextual factors, such as background of the analyst and complexity of the application, affect the time estimates, these estimates, at a minimum, show relative values for effort and training.

[2] Published reports (e.g., Nielsen, 1992; Karat, Campbell, & Fiegel, 1992) include other numbers as estimates of the time to perform some of these analyses, they are reporting actual times for specific, small design situations. The numbers here are intended to be more wide ranging, applying to more real-world design situations.

Define the Problem

Method	Effort	Training	References
Naturalistic Observation *(diaries, videotape, etc.)*	2 days	3 months	Hill, Long, Smith, and Whitefield, 1993
Interviews *(incl. focus groups, decision tree analysis, semantic nets)*	1 day	1 month	Ruddman and Engelbeck, this volume; Nielsen, Mack, Bergendorff, and Grischkowsky, 1986
Scenarios or use cases *(including envisioning)*	1 day	1 month	Jacobson, 1992; Carroll, in preparation for 1995
Task Analysis *(including operator function model)*	2 days	3 months	Johanssen, Hannum and Tessmer (1989)

A variety of methods center on detailing the tasks for which the system will be built. In some of the methods, the analyst either watches the users do their work, such as ***naturalistic observation*** and ***analysis of work practice.*** Data on these activities can be collected by being videotaping workers or by workers keeping ***diaries***. In all these methods, the analyst is interested in understanding the practicalities of how the work actually gets done in the current system, the details of the task requirements, the order in which the users do subtasks, what material/information they need, how they communicate with each other, how much time each subtask takes, and the physical, social and organization setting in which this work takes place.

Another set of ways of gathering requisite information involve asking users directly about their needs. ***Focus groups*** or one-on-one ***interviews*** are included in this set of methods. When the task to be supported is entirely new, potential users are encouraged to imagine the new setting and capabilities and are asked what they would do with it and how it would be used, a method called ***envisioning.***

Several new innovative methods are introduced in this volume. Interviews can elicit from the task performer the particular mental steps they go through, which can be represented in a ***decision tree.*** Ruddman and Engelbeck, this volume, for example, attempt to reconstruct the sequential cognitive processing that users perform in solving complex problems, such as configuring telephone service for home users in the face of a myriad of offerings and billing arrangements.

For those tasks that have a nomenclature that is complex and foreign to the analyst, ***semantic nets*** can represent the organization of objects and the terms used in the task domain. This technique (used by Ruddman and Engelbeck, this volume; see also Gillan and Breedin, 1990) complements the decision tree analysis, above, in that it uncovers complex data structures in the user's vernacular, whereas the decision tree analysis represents the processes that operate on that data. Knowledge engineering methods could potentially be applied to uncover the kinds of strategic activities and knowledge structures people use in domains that are unfamiliar to the designer, as suggested in Olson and Biolsi (1991).

Narrative descriptions of complete tasks are often called ***scenarios*** or ***use cases*** (McDaniel, Olson, and Olson, 1994).[3] While gathering these, there is also opportunity to elicit and discuss the problems users have noted with the current system and their wishes for the future.

Information from these observations can be represented in terms of ***activity*** or ***task analyses,*** where sub-tasks are coded and summarized into flow charts, frequency tables and state-transition diagrams (See for example, Sasso, Olson and Merten, 1987). These diagrams can then be used to suggest new ways of accomplishing the same goal. When measures of performance like time or quality are collected as well, these can be used to compare the old system with the new.

The ***operator function model*** (Mitchell, this volume) is a specific method for displaying a detailed task analysis. It makes explicit what information the operator/user needs at each moment in the task and the sequence of sub-tasks that the operator goes through (and the branches, the different things that could be done). This then can be used to suggest both what information must be displayed at each moment, what steps the user should be able to go through, and what options should be presented to the user in an easy-to-access command sequence. It is a relatively small step to go from this detailed specification to an actual prototype. Furthermore, it could well serve as the input to a rapid prototyping system such as ITS (Gould, Ukelson, and Boies, this volume).

---

[3]Use cases are essential components of the new Object Oriented Methods, now increasing in favor in the software design community (Jacobson, 1992).

Generate a Design

Method	Effort	Training	References
Building on Previous Designs *(steal and improve, design guidelines)*	1 day[4]	1 month	Perlman, 1988; Tetzlaff & Schwartz, 1991
Represent Conceptual Model	1 day	2 months	Moran, 1983
Represent Interaction *(GTN, Dataflow, etc.)*	2 days	2 months	Kieras and Polson, 1983
Represent Visual Displays	2 days	3 months	
Design Space Analysis *(QOC, Decomposition analysis)*	3 days	1 month	MacLean, et al, 1991

Design is essentially evolutionary. New designs borrow from and improve on previous designs. Even radically new designs are reactions to existing designs. This is a practical matter of the "reuse" of designs that already work and that users are familiar with. Thus, designers do not have to start from scratch and users do not have to learn yet again.

There are two distinct "modes" of borrowing: global and local. One can start by adopting the global models of existing systems. Card (this volume) calls this ***steal and improve.*** For example, there are many common concepts among word processing systems, among data base systems, and among spreadsheet systems. You would want to think hard about how and why you would want to be different if you were developing one of these kinds of systems. You can also adopt localized design components, such as user interface "widgets," many of which appear in toolkits. When the designer adopts a ***toolkit***, other design decisions are already made as well. For

[4] This may take longer. It requires the analyst to know the previous designs, for example, from competitive analyses, which normally take at least a week.

example, in the Macintosh toolkit, the designer has no choice as to how menus appear and how selections are made. And, embedded in the toolkit instructions are guidelines as to how to make some interface features match other systems of similar type, such as the guide to make Macintosh applications all have Apple, File, and Edit, as the leftmost three menu items.

Design guidelines exist for some significant portion of the components of a user interface, which are helpful to this initial design. For example, guidelines offer prescriptions about how to organize items on a set of menus, with ordering either by conceptual category (e.g., editing commands) or by frequency of use (e.g., Smith and Mosier, 1984). Other guidelines suggest the consistent placement of help and warning messages on the screen, and general use of Gestalt principles of proximity and dissimilarity to capture and guide the user's eye gaze to appropriate portions of the display.

While borrowing can often provide a starting place for design, especially in domains where there are existing systems, the goal of design is to create a system that addresses the particularities of the problem at hand. Generating a design involves creating descriptions of the design, and different kinds of design representations are essential for this activity.

A conceptual model is the set of conceptual entities that the system represents, that the user needs to understand to effectively use the system. It is important that the designer be clear about this and explicitly represent the conceptual model (Newman & Sproull, 1979). For example, understanding the hierarchical nature of text objects (characters, words, sentences, paragraphs, sections, chapters, documents) separately from the features of

layout (font, line, page, margin) helps construct appropriate actions and associated specifications. Moran (1983) proposed a representation of the relationship between the concepts in the users work domain (uncovered, say, by interviews or some form of semantic analysis) and the conceptual model embodied by the system. A useful specific version of this is an ***object/action*** analysis. For example, Moran used this kind of representation to assess the differences between line-oriented and full-screen editors, showing that the conceptual model of the former (strings of things to be replaced) required a translation from the conceptual model of the way we normally think of text.

In Object/Action analysis, the nouns and verbs of the domain and task are arrayed in a table. For example, in text editing, the table would show objects such as letters, words, sentences, paragraphs, pages, documents on one dimension, and actions such as copy and delete on the other. The table shows the complete matrix of commands that have to be offered. But the analysis shows additionally features of the domain model--how characters and sentences are to be treated differently from layout features like lines and pages These tables encourage completeness of actions on all objects, and consistency where appropriate. Such an analysis was one of the key design methods used in the construction of the Xerox STAR system, which resulted in the use of various "universal command keys", such as delete, copy, etc. (Smith, Irby, Kimball, Verplank, & Harslem, 1982).

Once the domain is known and the objects and actions specified, it is useful to represent dynamics of the interaction. ***State transition diagrams*** are the most common form of representation of interaction. They are networks of states (screens or modes) where all the possible actions that can be taken at

each state are drawn leading to the next states. A simple form of this is the diagram of connected menu items that some training manuals show as a summary of the system's functionality. A more sophisticated representation is the generalized transition network (Kieras and Polson, 1983), which accommodates the hierarchic nesting of contexts.

The commands and flow of the interaction are depicted in the representations described above. The ***visual display*** is best represented by renderings of the display on paper or other media. In PICTIVE, analysts give participants various "pieceparts" of interfaces made out of paper or plastic (e.g., menus, dialog boxes, scroll bars and windows) to arrange in an interface layout. When these pictures (full screenshots) are displayed on a board in the order in which they will appear, they are called a ***storyboard,*** similar to those used in the construction of films. The closer the representation to the final embodiment of what the user will see, the better the analyses of the arrangement, its attention getting abilities, and the ease of finding what the user needs to know at each moment.

Design involves discovering and evaluating many different possible designs--a "design space"--and also discovering the critical issues or questions that must be addressed. Often it is useful to keep track of the possibilities so that it is clear why particular design decisions are made. Formal analyses of these possibilities come from two methods, ***Design Space Analysis*** and ***Decomposition Analysis***. Both involve systematic processes by which various issues or questions about the design are raised and recorded, and then different alternatives, options, or solutions are offered and analyzed with well agreed-upon criteria. The Design Space Analysis takes many forms;

QOC (for Questions-Options-Criteria) serves as a good example (McLean, Young, Bellotti. and Moran, 1991). In QOC, the designer(s) systematically documents the questions to be addressed. Attached to each question are the various alternative solutions. Appropriate criteria (e.g., consistency, ease of programming, etc.) are then applied to each alternative, leading to a design decision. Decomposition Analysis is similar, except less formal and at a coarser grain (Olson, 1985). Here, the major components of the interface are the representation of the underlying data structure, the command entry style, the provision of memory aids, the access to help, etc. Each is examined in turn, alternatives generated, and then evaluated by any of a variety of means. In both of these methods, it can be seen that generating a new design and reflecting on it are tightly intertwined.

Reflect on the design

Method	Effort	Training	References
Checklists	1 day	1 week	Shneiderman, 1992
Walkthroughs	2 days	3 months	Lewis et al, 1990
Mapping analyses *(Task action, metaphor, consistency)*	2 days	2 weeks	Douglas and Moran, 1983; Payne and Green, 1986
Methods analyses *(GOMS, KLM, CPM, CCT)*	3 days	1 year	Card, Moran, and Newell, 1983; Kieras, 1988.
Display analyses	3 days	1 year	Lohse, 1991

Once we have some explicit representations of the design, there are a variety of ways to begin to assess the projected usability as well as the ease of

learning of this design. The methods listed here can also be used to some extent in generating designs. But these methods are detailed and require a fairly specific representation of a design to work on. They can be used either by analysts working alone, with users as part of the design team, or by analysts watching users trying to use a representation of the interface to perform some test task.

The quickest way to evaluate a first-cut or set of alternative components of the design involves answering a set of questions from a ***checklist*** (Shneiderman, 1992; Nielsen, 1993). Checklists and usability heuristics serve as memory aids to designers reminding them about prescriptions for the ease of learning and ease of use for each of the major components.

Just as programmers do a code walkthrough to check how the flow of the communication proceeds in the program and to check for things missing or conflicting, user interface designers perform a ***walkthrough*** of the user interface (Weinberg and Freidman, 1984). Here, however, the flow is from the user's perspective, where the user has goals in mind and tries to perform the actions necessary to accomplish those goals. With a good set of core tasks as test cases, a number of errors can be detected, especially in the flow of actions (whether they fit the order in which the user thinks of the actions) and in the availability of all the subfunctions needed.

Two variants of the walkthrough are the ***cognitive walkthrough*** (Lewis, Polson, Wharton, and Rieman, 1990; Rieman, Davies, Hair, Esemplare, Polson, and Lewis, 1991) and the ***claims analysis*** (Carroll and Rosson, this volume). Like the walkthrough above, they begin with the analysts generating

a core set of common tasks and the detailed step by step listing of what the user has to do to accomplish these. The analyses that follow, however, are much more explicit than in a standard walkthrough. The methods provide sets of questions the analyst is to ask about the interface. These questions are designed to highlight those aspects of the interface that are known to be difficult or error inducing. Claims analysis additionally encourages an explicit discussion of design tradeoffs, similar in style to the decomposition analysis, above.

Task-mapping analysis begins with a formal representation of the goal-action mapping that the user conceives of, and lays along side it the goal-action mapping that the system requires (Polson & Kieras, 1985). The analysis of the side-by-side diagrams shows mismatches which can turn into difficulty for the user to learn or execute. For example, a system that makes you move a range of text by selecting the range to be moved first, then the target then the action "move" mismatches the normal English command sequence that begins with the word "move" and continues by specifying the target material and the "move-to" location. Although today most wordprocessors follow the noun-verb format for commands, the novice in wordprocessing must *learn* that particular word order, since it is not fitting the order they have learned from spoken language.[5]

Similarly, Douglas and Moran (1983) suggest a careful ***analysis of the metaphor*** chosen for learning a new piece of software. They showed that by lining up the goal-action pairs of a target system (e.g., a text editor) and a

[5]This analysis, of course, depends on what you take as "natural." English imperatives may be verb-noun, but manually one first grabs a thing then does something with it, i.e., noun-verb.

metaphor system (e.g., a typewriter) there were a number of mismatches, some of which could significantly impede a new learner's understanding of the system. Since new learners will construct a metaphor or mental model even in the absence of instruction (Halasz and Moran, 1983), care should be taken to choose a helpful one and to teach it early.

Object/action analysis and ***state transition diagrams,*** described above, can have benefit in analyzing designs as well as in generating the first design. ***Analysis of the grammar*** of the command language, such as Moran's Command Language Grammar (CLG) (1981), Reisner's formal grammar (1984), and Payne and Green's Task Action Grammar (TAG)(1986), first represent the rules by which components form to produce a language of commands. Argument is made that the smaller the number of rules in the grammar, the easier the system will be to learn. The importance of these mapping analyses is that they focus the designer on the relationship between the elements of the system begin designed and the users' task domain and previous knowledge, rather than viewing the design in elegant but unnatural isolation.

Card, Moran, and Newell (1983) generated various ways to assess the moment-by-moment cognitive/perceptual/motor resources being used when interacting with a particular device, pioneering the field of cognitive engineering. The core idea is that for certain kinds of well-learned tasks, one could model the goals the user had, the methods offered by a system to satisfy these goals, the choices people made in varying circumstances, and the operator sequences that followed. From this model, called ***GOMS,*** and a related, more detailed model called the ***Keystroke Level Model,*** one can

fairly accurately predict how long a task will take (Olson and Olson, 1991). John (1988) introduced ***Critical Path Analysis*** that, in contrast to the GOMS model which assumes a sequential flow of cognitive processes (the recognize, retrieve, act cycle), recognizes that some tasks involve some cognitive processes that occur in parallel. This is often appropriate when the task to be modeled is performed repeatedly and rapidly in a high performance situation (see Atwood, Gray, and John, this volume).

Cognitive Complexity Theory, another important extension of the original cognitive engineering modeling of Card, Moran, and Newell, represents the knowledge needed to perform these tasks. Using this theory, Kieras and Bovair (1986) were able to predict how long a task would take to learn. All of these detailed analyses highlight the portions of the task that will take longer than necessary (e.g., too many things to remember or an overloaded working memory that generates errors), focusing redesign efforts to concentrate on very particular interaction details.

Several methods have arisen to assist in ***display analysis***. Tullis (1988) and Mackinlay (1986) developed programs to assess the crowding and thus readability of various aspects of a display. And, more recently, Lohse (1991) has constructed a perceptual simulation that will take a display as input and calculate how long it will take to answer a particular question about the display or to find certain critical features.

Build a Prototype

Method	Effort	Training	References

Prototyping tools	1 month	3 months	Wilson and Rosenberg, 1988
Participatory prototyping	1 week	2 months	Muller, 1991

The above methods analyze the plans or requirements of the system. They are conducted by the analyst, without direct involvement with the users of the system. To date, no one has found these analyses to be sufficient to find all the design difficulties (Nielsen, 1992; Karat, Campbell, Fiegel, 1992). All comparative studies of methods of design of user interfaces have found value to having users actually attempt to perform a realistic task using some form of the interface.

Also, although representations of designs produced in the initial generation are sufficient for the analysis of Reflection, they are not concrete. They can be understood only through the narrow lenses of the particular analyses. A concrete working prototype is needed in order to obtain rich empirical and experiential feedback.

The system used in these evaluations need not be the final system. The prototype can be presented effectively with ***paper, storyboards,*** and other media. One can produce sequences of screens similar to a movie production storyboard, or a complex book of printed screens whose sequencing is controlled by a human analyst. These allow rapid testing for flow of control, visual clarity, etc. without having to program a system to be fully operational. A variant of these simple prototypes is embodied in PICTIVE (Muller, 1991). When the end users put the requisite pieces of the interface together, it is called ***participatory prototyping*** (Poltrock and Grudin, this volume).

Although this kind of prototyping might work at this stage, it is more likely effective as a first cut that can then be further refined through analysis.

Toolkits provide easy, cost effective ways to construct a working interface for analysis and testing (Perlman, 1988; Hix and Schulman, 1991). ITS (Gould et al, this volume) discuss various ways to display the dialog design with various visual options.

Test the Prototype

Method	**Effort**	**Training**	**References**
Open Testing (*Storefront or hallway, alpha, and damage testing)*	1 week	1 month	Gould, et al, 1987
Usability Testing	2 weeks	1 year[6]	Gould, 1988

Once the system is mocked up using one of these methods, the users are then asked to work through a sample realistic task while the analysts collect various forms of data about users' performance. These can be reactions to attractiveness or appeal, ease of learning how to use it, or other characteristics of the user's ease in performing basic tasks. This method is variously called ***storefront*** or ***hallway testing,*** best exemplified in the design process used for the Olympic Messaging System (Gould, Boies, Levy, Richards, & Schoonard, 1987). In ***alpha testing,*** the prototype is given to associates, who then give feedback to the designers about usability; in ***damage testing,*** users deliberately try to break the system, giving feedback to the designers about the system's robustness.

[6]This estimate is for "delux" usability testing. "Discount" testing (Nielsen, 1992) is much faster to learn.

More formal analyses involve full-fledged ***usability tests*** in which users are taught the system (which itself provides an early test of the training materials) and asked to perform a set tasks. Early tests of the system often involve "critical tasks" which push the system and the user's capabilities so that they would be sure to see its fragile points. In other situations, when the goal is to find expected times to learn and perform, more conventional, common tasks are used, called "benchmark tasks."

A whole variety of measures are possible, including the time to learn, the time to perform particular tasks, individual keystroke times (for assessment of match to predictions from the Keystroke Level Model), error types and frequencies, thinking aloud (for assessment of goals and problem solving strategies), preference and satisfaction. The data from usability tests are relatively easy to collect; one can tell fairly quickly whether there are major design errors. More detailed comparison of moment by moment keystroke times with those projected from cognitive engineering allow designers to focus on those aspects that seem to present difficulties to the user. What is not easy is fixing these difficulties, especially since every design decision involves tradeoffs; each fix changes some other aspect, the overall change needing re-testing and/or analysis.

Implement the System

Method	**Effort**	**Training**	**References**
Toolkits *(e.g., Motif, NeXTstep, Apple)*	3 months	6 months	Perlman, 1988

The advantages of building the prototype in a full-scale toolkit center on the fact that the interface is not only easy to build, has style guidelines built in, and is relatively easy to change after usability testing, but toolkits generate production code, unlike prototypes built in some system like HyperCard. With toolkits, it is not necessary to rewrite the interface into the language of implementation.

Deploy the System

Method	Effort	Training	References
Internal testing	1 week	1 month	
Beta Testing *(logging, metering, surveys)*	2 weeks	1 month	Mackay, Malone, Crowston, Rao, Rosenblitt, and Card, 1989

Once a system is judged satisfactory, it is typically tested further first in the local environment, and then to an outside friendly environment, often a site that would like to be an early adopter of the technology in exchange for feeding back discovered bugs and mismatches in design. These tests are often called ***beta testing***. At this point, often data are collected, but of a less fine-grained sort. Two good sources are catalogued queries that come in on a help line and answers to questionnaires sent to customers or end users. It is also possible, in some situations, to log or meter the new system, just as one would do on an existing system, mentioned above. With keystroke data collected, for example, one can infer both what common tasks are being done efficiently or not, and overall use of system features.

What the Designer needs to Know to Choose a Method for the Right Time and the Right Task

The organization of the methods in the list above conveys that they apply to different activities of the development process. They also differ in the amount of time they require, the amount of detail uncovered and the accuracy of the conclusions that result. For example, using a checklist on the current or proposed design takes several hours and produces general recommendations about usability and learnability. The checklist can help determine which of two competing software packages might be easier for the end user, but will not provide enough detail to determine how long the task will take or what skill or domain knowledge the end user will have to have to behave accurately and with reasonable speed. In contrast, GOMS analysis and its partners, CCT and Keystroke Level Model, require a great deal of time, but provide detail necessary to say what users will have to know to perform the tasks well, roughly how long it will take to learn, and how long representative tasks will take to perform.

We also noted that the methods differ in how much the designer needs to know about human cognition, perception, and motor movement. Task analyses require very little such knowledge, as do some forms of checklists or guidelines and the cognitive walkthrough, whereas claims analysis, Cognitive Complexity Theory and Keystroke Level Model require a great deal.

That is, the methods differ in

the amount of **time** they take for and the concomitant level of detail and risk associated with the findings, and

the **knowledge** the analyst is required to have about basic cognitive processes of users.

Some methods are particularly suited for some kinds of users and tasks and not for others. This is probably the most difficult thing for the designer to assess. For example, tasks such as information retrieval, financial planning, piloting an airplane, and rapid transcription of text are very different in how sequentially and deliberately the user goes through the steps in the task. There have been numerous attempts to develop a task taxonomy (See for example Lenorovitz, Phillips, and Kloster, 1984 for a short review), but in general the taxonomies are far too detailed for the use we wish to put them to here. However, the analyst does need some guidance as to which of the methods suits the particular user's task, the one being designed.

For our purposes, the following seem to be the major dimensions on which a wide set of users' **tasks** differ:

1) the task is performed either as a set of sequential steps or as a . rapidly overlapping series of sub-tasks;
2) the task either involves high information content, with consequent complex visual displays to be interpreted, or it involves low information content, where simple signals are sufficient to alert the user that the next step is to proceed:

3) the task is intended to be performed either by a layman without much training or by a skilled practitioner in the task domain.

The first dimension has to do with whether the user's actions are deliberate and single minded, much like using a spreadsheet. This contrasts with tasks such as air traffic control, where attention rapidly shifts between input streams and goals are intertwined. Air traffic control similarly is high in information content, the second dimension, whereas the task of assigning a student a workstation in a computer lab is low in information content, much more like reacting to a simple signal (the request of the student). The third dimension reflects the assumed knowledge or skill level of the users. A bank teller machine has to be recognizable by any customer whereas a computer aided design (CAD) system is specific to a professional domian with its own shared vocabulary, and can be designed with the assumption that the designer will be trained in its use.

Most of the methods are applicable to both sequential and overlapping tasks. The one major exception is the GOMS/CCT family of models and accompanying analyses. They fit those tasks that are comprised of subsets of sequentially performed operators (either mental or motor). The Critical Path Analysis grew from this set of models to explicitly accommodate rapid-fire tasks that most likely involve cognitive/motor components that overlap in time. (See Atwood, Gray, and John, this volume).

When tasks are rich in information content, it is important to both determine the structure of the information as the user understands it, and to display it in a representation that visually maps well to that understanding. Therefore,

those methods that assess the organization of information objects and actions, the mental model of the system, and analysis of particulars of the perceptibility of visual displays are particularly relevant.

Interfaces for tasks that are designed for casual use by the layperson, that do not assume knowledge in a particular domain, should be assessed in particular for their learnability and the provision of information on the screen that suggests to the user what to do next. Several methods, such as the cognitive walkthrough, storefront analysis, and claims analysis, are particularly relevant for assessing this aspect of the interface.

If the task will be performed by a large workforce of dedicated users, then the more detailed methods, like GOMS, grammar analysis and Critical Path Analysis, will likely provide significant payoff. For example, there is a significant workforce that reconciles mismatches in customer claimed deposits and the accounting ledger in a bank "back room." These people work full days at a rapid pace. It is particularly important in this task that the information that the user needs to access to solve a problem be placed on the screen in tandem, and that the key information is readily readable. Careful analysis of the eye movements, clarity of font, and keystroke or command sequence is very important to a good design in this task, so that information is not lost out of the user's working memory, and that extra scans are not required to "line up" the aspects of the accounting that mismatch. Good screen design can shorten each reconciliation task by seconds. Although mere seconds are saved, when multiplied by the number of tasks accomplished per day and the number of operators doing such a task, the savings could accrue to millions of dollars.

Some of the methods, like the GOMS, CCT, Keystroke and grammar analysis require weeks to do for any medium size task and system. They are very detailed, cataloging not only the action steps of the potential user, but the cognitive/perceptual/motor steps as well. They provide, however, a great amount of detail. They are therefore only appropriate when that kind of investment in time will reveal important details of the speed of interaction or complexity that might produce significant errors. They have shown value in situations where new operator workstations are being designed for high-speed work (Atwood et al) and for situations where errors are very costly, such as wrong business decisions caused by the wrong data being retrieved from a large data base because of its complex user interface (e.g. Smelcer, 1989).

These time-consuming methods often also require detailed knowledge about cognition. GOMS family of models and methods require the analyst to know facts about when in a task information might reside in short-term memory and how far an eye movement will jump on a visual display of certain size. Even the claims analysis requires intuition about these processes to help discover what the appropriate and inappropriate claims are that the artifact embodies. In contrast, methods like checklists and walkthroughs often can be conducted by people without an intimate knowledge of cognition and perception, and are therefore at the same time faster to accomplish and less accurate. User testing often takes several weeks to accomplish (including building the prototype, watching the users and analyzing the results), but can be done by careful but not necessarily trained observers.

Summary of Costs and Benefits

To provide guidance to the designer, we have prepared a table that highlights four characteristics:

The methods are of different **types**. Some collect data (empirical), some are analyses of the structure of the task and interface (analytic), and some construct various representations of the interface (constructive).

The **benefits** of the method in terms of what aspect of the interface that the method is particularly suited to reveal ---the task steps, the performance or learnability, or the user's acceptance of the system (called tasks, perform, learn, or accept in the table)

Two aspects of the **costs** of using the method--the **effort** to use it (which often correlates with amount of detail) and the **training** needed.

This table provides a rough assessment of these characteristics. It was constructed and synthesized by the authors guided by input from the members of the workshop at Boulder. This table is intended to be advisory about the occasions when the method will or will not be useful.

Examples of Coordinated Use of the Methods

Table 2 provides some guidance to the selection of methods for the particular design task at hand. But, just as good cookbooks give not only selection criteria for individual dishes but also suggest combinations of dishes to create a pleasing meal, we provide here two such "meals." The first illustrates the use of quick methods for a simple walk-up-and-use system, such as an ATM. The second illustrates the set of design methods at the other end of the

continuum, where the interface is information rich, and speedy, accurate real-time performance is critical to the operator's success.

The literature contains several other descriptions of a coordinated sets of methods. Gould's description of the development of the Olympic Messaging System (Gould et al, 1987) demonstrates the coordinated use of several methods for walk-up-and-use interfaces, and the description by Ruddman and Engelbeck (this volume) about the development of an interface for the operator's support for configuring new telephone service demonstrates coordinated methods for an information rich, interactive system involving customer conversation. McDaniel, et al (1994) describe the use of a combination of HCI methods, those from Business Process Redesign (Hammer and Champy, 1993) and those in Object Oriented methodology (Jacobson, 1992) in the design of a system to help space physicists access remote sensors and to converse about them across several continents (McDaniel, Olson, and Olson, 1994).

Table 2. Summary of Costs and Benefits of the Methods

Method	Types	Benefits	Costs–Effort	Costs–Training
DEFINE THE PROBLEM:				
Naturalistic Observation *(diaries, videotape, etc.)*	empirical	tasks	2 days	3 months
Interviews *(incl. focus groups, decision tree analysis, semantic nets)*	empirical	tasks	1 day	1 month
Scenarios or use cases *(including envisioning)*	analytic	tasks	1 day	1 month
Task Analysis *(incl. Operator Function Model)*	analytic	tasks	2 days	3 months
GENERATE A DESIGN:				
Building on Previous Designs *(steal and improve, design guidelines)*	constructive	tasks, perform, learn, accept	1 day	1 month
Represent Conceptual Model	constructive	learn	1 day	2 months
Represent Interaction *(GTN, Dataflow Diagram)*	constructive	perform, learn	2 days	2 months
Represent Visual Display	constructive	perform, learn	2 days	3 months
Design Space Analysis *(QOC, Decomposition Analysis)*	analytic	tasks, perform, learn	3 days	1 month
REFLECT ON THE DESIGN				
Checklists	analytic	perform, learn	1 day	1 week
Walkthroughs	analytic	perform, learn	2 days	3 months
Mapping analysis *(task-action, metaphor, consistency)*	analytic	perform, learn	2 days	2 weeks
Methods analysis *(GOMS, KLM, CPM, CCT)*	analytic	perform, learn	3 days	1 year
Display analyses	analytic	perform, learn	3 days	1 year
BUILD A PROTOTYPE:				
Prototyping tools	constructive	testable system	1 month	3 months
Participatory prototyping	empirical	tasks, accept	1 week	2 months
TEST THE PROTOTYPE:				
Open Testing *(Storefront or hallway, alpha, damage testing)*	empirical	perform, learn, accept	1 week	1 month
Usability Testing	empirical	perform, learn, accept	2 weeks	1 year
IMPLEMENT THE SYSTEM:				
Toolkits *(e.g., Motif, NeXTstep, Apple)*	constructive	fully testable system	3 months	6 months
DEPLOY THE SYSTEM:				

Internal testing	empirical	perform, learn, accept	1 week	1 month
Beta Testing *(logging, metering, surveys)*	empirical	tasks, perform, learn, accept	2 weeks	1 month

Coordinated methods for quick evaluation of a walk-up-and-use system. An ATM is an example of a system which

> has simple sequential task flow (which presents information on choices the user has, each of which leads to new choices),
>
> is relatively low in information content (mainly verbal selections, for example about withdrawal or deposit and how much)
>
> and is targeted for the layman.

Because the task is performed by untrained users at their own pace, the emphasis is on the ease with which the user can learn to operate the device. Obviousness of what action to take next, and error recovery are prime. Also, since the business objective of this system is not rapid performance of tasks, but rather widespread use leading to offloading clerical tasks to the customer, the budget for construction and evaluation are likely small. Fast methods will do and the designer should not be expected to have a Ph.D. in cognitive psychology.

To discover the components of the task, simple **questionnaires** might suffice, asking the potential customers what kinds of choices they might be interested in. Often marketing has the basics of this information collected already, and use of this for the interface objects, actions, and flow will serve well. Since lots of ATMs exist already, it would also be appropriate to do some **naturalistic observation**, where designers observe current users at existing machines.

The initial design, guided by prescriptions from **guidelines** and assessed quickly with **checklists**, might be printed on paper, and displayed as a **storyboard**, for analysis of flow, screen display, etc. without users. Designers can view the storyboard for aspects of ease of learning, using in particular a **cognitive walkthrough**. The flow of the system could be assessed with a simple **generalized transition network**, to assure consistency in the use of error recovery and navigation commands.

For **hallway testing**, a mock-up of the entire display might be constructed, with a **rapid prototyping system** (such as HyperCard) embodying the design. Designers can observe the test users' difficulties, or get them to think out loud while they attempt to use the system.

After several short such analyses and redesigns, the system, in its penultimate form, can be installed at a friendly test site and some basic **system monitoring** data collected for analysis of gross usability and preference characteristics.

Coordinated methods for detailed evaluation of a system for high performance for dedicated, skilled users. At the other end of the spectrum is a system that supports back-room workers at a bank who are reconciling the machine-read check register with what the customer wrote on the back of a deposit slip. The task

requires the overlapping activation of the user's mental and physical capabilities (scanning the next set of materials while the previous tasks' corrections are keyed in)

is relatively high in information content (side-by-side displays of the deposit slip's handwritten entry and a list of the checks accompanying the deposit, both machine read and scanned in true copy)

and is targeted for the dedicated skilled user.

Because the task is performed all day every day by skilled users, there is considerable payoff from having detailed, somewhat time consuming analyses. The outcome has to be detailed enough to recommend changes to the interface that may bring about seconds of savings in each task completion. But, because of the high volume of performance of each task, savings accrue rapidly. Budget for construction and evaluation of this kind of system can be quite large, given the anticipated payoff.

To understand the task, which in this case is not particularly obvious to the system designer, several discovery techniques should be employed. If there is a previous system in place (check balances are reconciled in *some* way before this new system is built), the designers can engage in some **natural observation**, plus interview the workers about aspects that are difficult or annoying. More detailed discovery of the objects/actions of the task domain and the kinds of thinking that goes on during the execution of the task can come from **semantic net interviews**, **decision tree interviews**, and other

techniques from the area of knowledge engineering. A detailed **task analysis** is performed next, showing the order of subtasks, and the kinds of information that are needed at each moment so the user can perform requisite cognitive tasks to accomplish the goal. The task analysis may take the form of a **operator function model** with details of the knowledge necessary in the form of a **GOMS model** or some parts of the **Task Action Grammar**.

Once the task is fully understood, a series of design and evaluative iterations follow. The system can be generated and displayed as a **Generalized Transition Network** or a working prototype, using one of the more sophisticated toolkits like **ITS**. This design is then analyzed in detail for the cognitive and motor movements required to accomplish the task, using the **Critical Path Analysis** (CPA) variant of the GOMS family of models. Since the system has high information content, detailed analysis of the **visual display** is also warranted. **Usability tests** follow, with particular emphasis on the fit of the CPA to the actual timing of the task, to hone both the model's accuracy and to show where the system does not fit the predictions of optimal performance afforded by the model. The design iterates until the users' performance meets preset target criteria for skilled performance.

Discussion

The synthesis and table above are intended to be helpful, not to provide a detailed critique of each method for its usefulness. There are methods, for example, that have the goal of being useful and usable by designers, but at present are in a form that makes them difficult to learn or awkward to use. For example, one of the motivations for providing the Cognitive Walkthrough

was to make knowledge that is gleaned from GOMS and other empirical investigations accessible in a method usable by designers.

Also, the table format masks the potential synergy between methods, those useful links that can occur between methods. For example, the Operator Function Model is a detailed analysis of the object, actions, and flow of control necessary for task performance. Upon inspection, we discovered that its outputs are exactly what are needed for input to ITS. Thus, while one method might be effective only on its own, others may have useful links between them.

It is also the case that the table makes only crude assessments of the kinds of tasks that it can be applied to and coarse grained estimates of how much effort it takes and how much training the designer has to have in order to use the method successfully. Of particular concern is the implication that those methods that take a short time but give you broad coverage of evaluation are higher on the cost/benefit curve and therefore more valuable. Some of the methods, such as GOMS, although they take a long time to specify, have a large payoff for several different aspects of the design process. For example, once the task is specified in a GOMS terminology, not only is it possible to estimate how long a task will take (by using parameters in the Keystroke Level Model) but you also have the basic information necessary to write effective documentation (Gong and Elkerton, 1990). The GOMS model forces the analyst to understand the major tasks and the recommended procedure to accomplish those tasks, the basic elements of writing good training material.

Also, a listing of methods like that above misses some of the more global design process principles that successful designers offer. For example, it has

been widely recognized that an effective management procedure for assuring adequate attention to the user interface is to incorporate metrics of user acceptability into the same set of metrics that software designers are used to having to determine if the performance of the software itself is acceptable. (Good, Whiteside, and Wixon, 1984). There are other principles for effectively using the methods in the design process. Having the software developers themselves on the team that runs a usability evaluation is recommended because they can see for themselves in real time that aspects of a current design provoke repeated difficulties across users. Summary reports do not convey the same weight for such conclusions as do real-time experiences. And, it has long been recommended that users themselves sit on the design team, to assure adequate input of task vocabulary, completeness of features, and flow that fits the way the user thinks about the task progression. Many of the methods listed above could benefit from users being on the design or analysis team.

This overview shows that there is considerable progress in providing ways to design useful, usable, and learnable user interfaces. Many new methods have been developed since the 1983 NRC report, and recent studies have compared the cost/benefit of various methods (Nielsen, 1992, 1993). We have provided a framework for seeing the roles of different methods, but more work is needed on a detailed cost/benefit of the methods. Not only do the methods need to be assessed for their usefulness, but new methods need to be developed that are more complete and usable.

Acknowledgment. This work has been supported in part by a grant from Army Research Institute, contract number MDA 903-89-K-0025.

References

Adams, J. L. (1979) *Conceptual Blockbusting: A Guide to Better Ideas.* NY: W. W. Norton and Company.

Anderson, N., and Olson, J. Reitman (Eds.) (1985) *Methods for Designing Software to Fit Human Needs and Capabilities: Proceedings of the Workshop on Software Human Factors.* Washington, D. C.: National Academy Press.

Atwood, M., Gray, W. John, B. Project Ernestine: From basic research to application and back again, and again, and again...(this volume)

Card, S. (this volume) *so far, just listed as "discussant"*

Card, S. K., Moran, T. P., and Newell, A. (1983) *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Carroll, J., and Rosson, M. B. (this volume) Getting around the task-artifact cycle: How to make claims and design by scenario.

Douglas, S. and Moran, T. (1983) Learning text editor semantics by analogy. *CHI'83 Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM. pp.

Gillan, D. J., and Breedin, S. D. (1990) Designers' models of the human-computer interface. *CHI'90 Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM. pp.

Gong, R., and Elkerton, J. (1990) Designing minimal documentation using a GOMS model: A usability evaluation of an engineering approach. Proceedings *of the Conference on Human Factors in Computing Systems.* NY: ACM. 99-106.

Good, M., Whiteside, J. Wixon, D., and Jones, S. (1984) Building a user-derived interface. *Communications of the ACM,* 27, 1032-1043.

Gould, J. D., (1988) How to design usable systems. in M. Hellander (Ed.) *Handbook of Human Computer Interaction.* Amsterdam: North Holland, 757-785.

Gould, J. D., Boies, S. J., Levy, S., Richards, J. T., and Schoonard, J. (1987) The Olympic Messaging System: A test of behavioral principles in system design. *CACM* 30, 758-769.

Gould, J., Ukelson, J. P, and Boise, S. (this volume) ITS: An emerging Methodology for user interface design

Halasz, F. G. and Moran, T. P. (1983) Mental models and problem solving in using a calculator *Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM, 212-216.

Hammer, M., and Champy, J. (1993) *Reengineering the Corporation: A Manifesto for Business Revolution.* New York: HarperCollins Publishers, Inc.

Hill, B., Long, J., Smith, W., and Whitefield, A. (1993) Planning for multiple task work - an analysis of a medical reception worksystem. *Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM, 314-320.

Jacobson, I. (1992) *Object-Oriented Software Engineering.* Reading, MA: Addison Wesley Publishing Co.

Jonassen, D. J., Hannum, W. H., and Tessmer, M. (1989) *Handbook of Task Analysis Procedures.* New York: Praeger.

John, B. E. (1988) *Contributions to engineering models of human-computer interaction* Dissertation, Carnegie Mellon University Department of Psychology.

Karat, C. M., Campbell, R., and Fiegel, T. (1992) Comparison of empirical testing and walkthrough methods in user interface evaluation. *Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM., 397-404.

Kieras, D. (1988) Towards a practical GOMS model methodology for user interface design.

Kieras, D. E., and Bovair, S. (1986) The acquisition of procedures from text: A production-system analysis of transfer of training. *Journal of Memory and Learning*, 25, 507-524.

Kieras, D. , and Polson, P. G. (1983) A generalized transition network representation for interactive systems. *Proceedings of the Conference on Human Factors in Computing Systems*. NY: ACM, 103-106.

Lenorovitz, D. R., Phillips, M. D., Ardrey, R. S., and Kloster, G. V. (1984) A taxonomic approach to characterizing human computer interfaces. In G. Salvendy (Ed.) *Human Computer Interaction*. Amsterdam, The Netherlands: North Holland. 111-116.

Lewis, C., Polson, P., Wharton, C., and Rieman, J. (1990) Testing a walkthrough methodology for theory-based design of walk-up-an-use interfaces. *Proceedings of the Conference on Human Factors in Computing Systems*. NY: ACM, 235-241.

Lohse, J. (1991) A cognitive model for the perception and understanding of graphs. *Proceedings of the Conference on Human Factors in Computing Systems*. NY: ACM. pp. 137-144.

Mackay, W. E., Malone, T. W., Crowston, K., Rao, R., Rosenblitt, D. and Card, S. K. (1989) How do experienced users use rules? *Proceedings of the Conference on Human Factors in Computing Systems*. NY: ACM, 211-216.

Mackinlay, J. (1986) Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5 (2, April), 110-141.

MacLean, A., Young, R., Bellotti, V. M., and Moran, T. P. (1991) Questions, options, and criteria: Elements of a design space analysis. *Human Computer Interaction*, 6, 201-250.

McDaniel, S. E., Olson, G. M., and Olson, J. S. (1994) Methods in search of methodology--Combining HCI and Object Orientation. *Human Factors in Computing Systems: CHI'94 Conference Proceedings.* New York: ACM.

Mitchell, C. (this volume) Cognitive engineering models: A prerequisite to the design of human-computer interaction in complex dynamic systems

Moran, T. (1983) Getting into a system: External-internal task mapping analysis. *Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM, 45-49.

Moran, T. (1981) The Command Language Grammar: A representation for the user interface of interactive computer systems. *International Journal of Man-Machine Systems. 15,* 3-50.

Muller, M. J. (1991) PICTIVE-An exploration in participatory design. *Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM. pp. 225-231.

Newman, W. M., & Sproull, R. F. Principles of interactive computer graphics (and ed.). New York: McGraw-Hill, 1979.

Nielsen, J., Mack, R. L., Bergendorff, K. H., Grischkowsky, N. (1989) Integrated software usage in the professional work environment: Evidence from questionnaires and interviews. *Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM, 162-167.

Nielsen, Jakob (1992) Finding usability problems through heuristic evaluation. *Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM., 373-380.

Nielsen, Jakob (1993) *Usability Engineering.* Boston, MA: AP Professional.

Olson, J. S. (1985) Expanded design procedures for learnable, usable interfaces. *Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM. pp. 142-143.

Olson, J. S., and Biolsi, K. J. (1991) Techniques for representing knowledge. in Ericsson, A. and Smith, J. (Eds) *Toward a general theory of expertise.* Cambridge, England: Cambridge University Press.

Olson, J. S., and Olson, G. M. (1991) The growth of cognitive modeling since GOMS. *Human Computer Interaction*, 5, 221-266.

Payne, S. J., and Green, T. R. G. (1986) Task-action grammars: A model of the mental representation of task languages. *Human Computer Interaction,* 2, 93-133.

Perlman, G. (1988) Software tools for user interface development. In M. Hollander (Ed.) *Handbook of Human Computer Interaction.* Amsterdam, The Netherlands: North Holland. 819-833.

Polson, P. G., and Kieras, D. E. (1985) A quantitative model of the learning and performance of text editing knowledge. *Human Factors in Computing Systems, Proceedings of the CHI '85,* NY: ACM.

Poltrock, S., and Grudin, J. (this volume) Participant observer studies of interface design and development.

Reisner, P. (1984) Formal grammar as a tool for analyzing ease of use: Some fundamental concepts. in J. Thomas, and M. Schneider (Eds.) *Human Factors in Computer Systems.* Norwood, NJ: Ablex.

Rieman, J., Davies, S., Hair, D. C., Esemplare, M., Polson, P. and Lewis, C. (1991) An automated cognitive walkthrough. *Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM, 427-428

Ruddman, S. E., and Engelbeck, G. (this volume) Combining four task analysis approaches for the design of a complex user interface for telephone service negotiation

Sasso, W., Olson, J. S., and Merten, A (1987) The practice of office analysis: Objectives, obstacles, and opportunities. *Office Knowledge Engineering*, 2, 11-24.

Schon , D.A. (1983) *The Reflective Practitioner: How Professionals Think in Action.* New York: Basic Books.

Shneiderman, B. (1992) *Designing the user interface: Strategies for effective human-computer interaction.* Second edition. Reading, MA: Addison-Wesley.

Simon, H. A. (1981) *Sciences of the Artificial* Cambridge, MA: MIT Press.

Smelcer, J. B. (1989) *Understanding user errors in database query.* Unpublished doctoral dissertation, University of Michigan, Ann Arbor.

Smith, D. C., Irby, C., Kimball, R., Verplank, B., and Harslem, E. (1982) Designing the Star user interface. *Byte* 7(4) 242-282.

Smith, S. L., and Mosier, J. N. (1984) Design guidelines for user-system interface software. Mitre Corporation Report ESD-TR-84-190. Bedford, MA: Mitre Corporation.

Tetzlaff, L., and Schwartz. (1991) The use of guidelines in interface design. *Proceedings of the Conference on Human Factors in Computing Systems.* NY: ACM, 329-334.

Tullis, T. S. (1988) Screen design. in M. Hollander (Ed.) Handbook of Human Computer Interaction. Amsterdam, Netherlands: North Holland. pp 377-411.

Weinberg, G. M., and Freidman, D. P. (1984) Reviews, walkthroughs, and inspections. *IEEE Transactions on Software Engineering* SE-10(1).

Wilson, J., and Rosenberg, D. (1988) Rapid prototyping fore user interface design. in M. Hollander (Ed.) *Handbook for Human-Computer Interaction.* Amsterdam: North Holland. 859-876.